NotasIV Documentation

Angel Rodriguez Hodar

Dec 25, 2019

Indice:

Aqui se documentará todo lo relativo al microservicio.

CHAPTER 1

Herramientas

En esta sección se detallan las herramientas usadas para la creación y gestión del proyecto:

- `Lenguaje`: Se programará en **Python** debido a su facilidad de uso y cantidad de librerias disponibles, permitiendo un mayor foco en la infraestructura que en las peculiaridades del lenguaje. En concreto se usará la versión **3.6.8**, ya que es estable pero no la última disponible, por lo que se pueden hacer pruebas de actualizar el entorno virtual y resolver posibles problemas de compatibilidad que se presenten.
- `Entorno virtual y gestion de librerias`: Una vez elegido el lenguaje, y mas siento Python, se va a usar la herramienta pipenv para gestionar el entorno virtual de ejecución y las librerias con las versiones necesarias de cada una. He elegido esta herramienta porque unifica el gestor de paquetes *pip* y el módulo *venv* de Python en una sola, además de ofrecer posibilidad de crear builds deterministas y usar el formato que sustituirá al famoso requirements.txt, el Pipfile
- `Framework Web`: Para interactuar con el microservicio se usará una API REST, por lo que utilizaré el famoso microframewok Flask para el desarrollo del apartado web, en concreto una extensión de la libreria llamada Flask-RESTplus que facilita la creación y el uso de buenas prácticas para la creación de la misma.
- `Tests`: Python dispone de varias librerias de testing. Las mas famosas son *unittest* (standard library), *nose* y *pytest*. De entre ellas he elegido **pytest** debido a la cantidad de extensiones (315+, como por ejemplo para coverage, facilidad de uso y perfecta retrocompatibilidad con las otras librerias mencionadas.
- `Almacenamiento`: Para almacenar los datos, he optado por utilizar **NoSQL**, en concreto **MongoDB** junto con el ORM mongoengine. He optado por esta vía porque a finales de verano estuve probando a usarlo en un proyecto y lo encontré muy facil de usar y gestionar, a la vez que me sirve para aprender nuevas tecnologías ya que yo al menos en cursos pasados de la carrera solo he dado SQL.
- `Logs`: En principio, usaré la librería loguru, la cual es una abstracción y mejora de la librería logging que viene incorporada en la standard library de python y que, según su autor, pretende mitigar los inconvenientes de la misma. Luego se integrará con el Elastic Stack para poder visualizar los logs comodamente a través de un dashboard (tengo que investigar mas de esto).
- `CI`: Para mantener un flujo de trabajo de integración continua, optaré por el uso de **TravisCI** dada su popularidad, aunque aun no he investigado mucho lo que ofrecen otros servicios como CircleCI como para decantarme por Travis al 100%.

CHAPTER 2

Construcción

Como herramienta de construcción se ha usado un `Makefile` ubicado en la raíz del proyecto, que contiene el siguiente codigo:

```
.PHONY: tests docs clean
init:
    pip install pipenv
    pipenv install --dev
pm2:
    sudo apt update
    sudo apt install -y nodejs
    sudo apt install -y npm
    sudo npm install -g pm2
tests:
    pipenv run python -m pytest -p no:warnings --cov-report=xml --cov=notas tests/
coverage:
    pipenv run codecov
docs:
    cd docs && pipenv run make html
start:
    pipenv run pm2 start "gunicorn -b 0.0.0.0:$(PORT) app:app" --name app
start-no-pm2:
    pipenv run gunicorn -b 0.0.0.0:$(PORT) app:app
stop:
    pipenv run pm2 stop app
delete:
    pipenv run pm2 delete app
restart:
    pipenv run pm2 restart app
heroku:
    sudo snap install heroku --classic
    heroku login
    heroku create notas-iv --buildpack heroku/python
    git push heroku master
heroku-docker:
```

(continues on next page)

(continued from previous page)

```
    sudo snap install heroku --classic
    heroku login
    heroku create notas-iv
    heroku stack:set container
    git push heroku master
docker-build:
    docker build -t notas-iv .
docker-run: docker-build
    docker run -e PORT=$(PORT) -p 5000:$(PORT) notas-iv
vm:
    vagrant up --no-provision
provision:
    vagrant provision
clean:
    rm -f coverage.xml .coverage
    cd docs && make clean
```

A continuación se explica el funcionamiento de cada regla:

- `init`: Instala el gestor de paquetes y entorno virtual pipenv, a la vez que instala las dependencias del proyecto.
- `pm2`: Instala npm y el paquete pm2.
- `tests`: Ejecuta los tests y genera un reporte en formato xml.
- `coverage`: Utiliza el reporte generado previamente para actualizar la página en codecov.io
- `docs`: Compila la documentación generando un directorio `docs/_build` con los archivos html para abrirlos con un navegador web.
- `start`: Inicializa un contenedor de pm2 con un servidor WSGI.
- `start-no-pm2`: Inicializa un servidor web WSGI sin usar pm2.
- `stop`: Para el proceso de pm2 (pero no lo borra de memoria). Si se ejecuta `start` posteriormente se reactiva ese proceso parado.
- `delete`: Para el proceso de pm2 y también lo borra de memoria.
- `restart`: Reinicia el proceso de pm2.
- `heroku`: Lleva a cabo todo lo necesario para desplegar la aplicación en Heroku. Para mayor información consulta la sección *Despliegue en un PaaS*
- `heroku-docker`: Lleva a cabo todo lo necesario para desplegar la aplicación en Heroku usando docker. Para mayor información consulta la sección *Docker*
- `docker-build: Crea una imagen de docker usando el ``Dockerfile`.
- `docker-run`: Ejecuta un contenedor con nuestra imagen (si no está creada la crea en ese momento).
- `vm`: Crea una máquina virtual con Vagrant. Para más información consulta la sección *Creación y aprovisionamiento*
- `provision`: Aprovisiona la máquina virtual creada previamente. Para más información consulta la sección *Creación y aprovisionamiento*
- `clean`: Limpia los archivos generados para codecov (útil para cuando se ejecutan en local y no en Travis por ejemplo).

Hay una directiva extra llamada `.PHONY` que evita confundir reglas con directorios existentes, como por ejemplo los tests. Para ejecutar la herramienta de construcción con los tests simplemente debemos hacer lo siguiente:

```
$ make
$ make tests
```

CHAPTER 3

Integración Continua

Como sistemas de integración continua se han usado TravisCI y CircleCI.

3.1 TravisCI

El sistema de Travis se configura únicamente con un archivo `.travis.yml` que debe estar ubicado en la raiz de nuestro proyecto. Este archivo contiene la siguiente información:

```
language: python
python:
  - "3.5"
  - "3.6"
  - "3.7"
  - "3.8"
  - "3.8-dev"
before_install:
  - make pm2
install:
  - make
script:
  - make tests
  - make start
  - make delete
after_success:
  - make coverage
```

Simplemente le definimos el lenguaje de programación que vamos a usar, junto con las distintas versiones del mismo con las que vamos a testear nuestra aplicación. Luego, antes de instalar las dependencias de nuestro proyecto, es necesario instalar npm y el paquete pm2 del mismo, y para ello hay que usar la regla before_install.

Una vez hecho esto, para ejecutar nuestra herramienta de construcción en el entorno que nos da travis primero debemos usar `make` para instalar las dependencias del proyecto en cuanto a librerias de python, en la regla install. Luego tan solo necesitamos escribir las reglas de nuestra herramienta de construcción para pasar tests, arrancar y parar el

microservicio. Por lo tanto, en la directiva `script` solo debemos ejecutar `make tests`, `make start` y `make delete`.

Si todo ha ido bien, usando la directiva `after_success` se ejecutará `make coverage` para mandar los reportes generados en la ejecución de los tests a la plataforma codecov.io.

3.2 CircleCI

Para CircleCI la configuración es bastante similar. En este caso el archivo de configuración pasa a llamarse `config.yml` y hay que ubircarlo en un directorio `.circleci` en la raiz de nuestro proyecto. El archivo `config.yml` contiene lo siguiente:

```
version: 2
workflows:
  version: 2
  test:
    jobs:
      - python-3.5
      - python-3.6
      - python-3.7
      - python-3.8
jobs:
  python-3.5: &build-template # Definimos una base de ejecución
    docker:
      - image: circleci/python:3.5
    steps:
      # Obtenemos codigo del repo
      - checkout
      # Entorno virtual y dependencias
      - run:
          name: Entorno y dependencias
          command: |
            make pm2
            make
      # Ejecucion de los tests
      - run:
          name: Ejecutar tests
          command: |
            make tests
      # Actualiza codecov
      - run:
          name: Coverage
          command: |
            make coverage
      # Arranca el microservicio
      - run:
          name: Arranque
          command: |
            make start
      # Para y borra el microservicio
      - run:
          name: Parada
          command: |
            make delete

  python-3.6:
```

(continues on next page)

(continued from previous page)

```
    <<: *build-template
    docker:
      - image: circleci/python:3.6
  python-3.7:
    <<: *build-template
    docker:
      - image: circleci/python:3.7
  python-3.8:
    <<: *build-template
    docker:
      - image: circleci/python:3.8
```

En este caso, aunque la configuración es menos trivial que con Travis, ya que por ejemplo para indicar la versión de python específica que queremos debemos buscar cual es la imagen de docker que contiene exactamente la versión que queremos. Aun asi, realmente es bastante intuitivo, permitiendo múltiples configuraciones y posibilidades.

CHAPTER 4

Clase

En esta sección se muestra la funcionalidad de los métodos de la clase que se va a implementar.

notas.clase.**delete_student**(*student_id*)
Removes a student

Parameters **student_id** (*str*) – The github username of the student.

Returns All students without the deleted one

Return type (dict)

notas.clase.**get_student**(*student_id*)
Returns a student by its given id

Parameters **student_id** (*str*) – The github username of the student

Returns Student data

Return type (dict)

notas.clase.**get_students**()
Returns all the stored students

Returns Dict of students

Return type (list)

notas.clase.**insert_student**(*student*)
Inserts a new student

Parameters **student** (*dict*) – Student data.

Returns All students with the new one inserted.

Return type (dict)

notas.clase.**update_student**(*student_id*, *field*, *value*)
Returns a student by its given id

Parameters

- **student_id** (*str*) – The github username of the student.
- **field** (*str*) – The field to update.
- **value** (*str*) – The new value for the field.

Returns All students with the selected one updated.

Return type (dict)

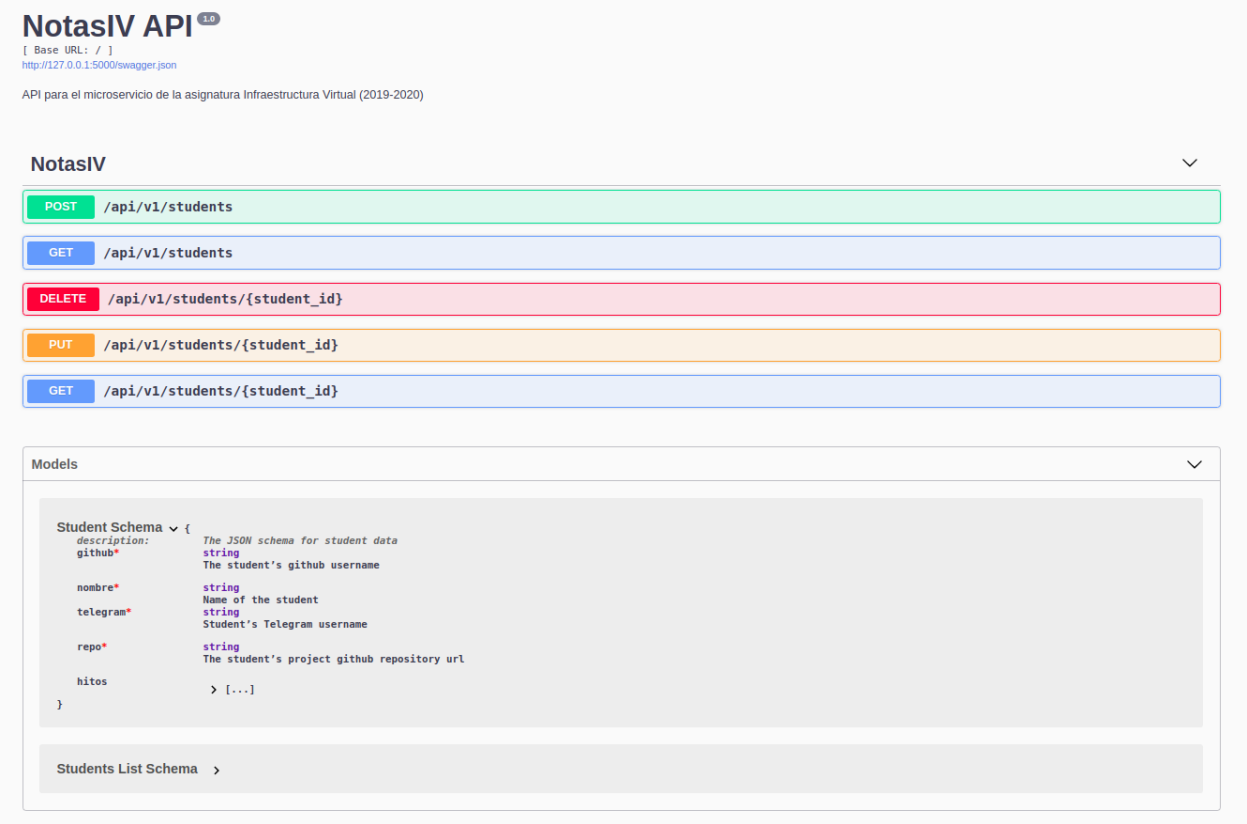
CHAPTER 5

API

En esta sección se muestra la documentación de la API implementada con Swagger para no solo ver los métodos que ofrece el microservicio, sino también para poder testearlo, así como la documentación de cada función de los tests.

5.1 Swagger

El paquete `flask-restplus` usado para el desarrollo de la API-RESTful, poniéndole una serie de decoradores a los métodos, genera una documentación con Swagger que puede ser visualizada en el endpoint `/` de la API con Swagger UI y muestra tanto los métodos disponibles como los modelos JSON devueltos por la API (aumenta el zoom en el navegador si no lo ves bien).



Swagger UI también ofrece la posibilidad de probar la API cómodamente como si uśaramos Postman por ejemplo. Nos dice los outputs que ofrece un endpoint y la posibilidad de hacer POST y PUT adjuntando cómodamente un JSON en el body de la petición. Para poder probar toda esta funcionalidad, como ya tengo la app corriendo en Heroku, puedes acceder a ella simplemente haciendo click aquí

5.2 Tests

Toda la funcionalidad referente a los tests sobre la API la puedes encontrar en el archivo `tests/tests_api.py`. A continuación se muestra una breve documentación sobre cada método implementado.

`test_api.`**`test_delete_student`**(*client*)

Testea si se recibe código 204 en la ruta /api/v1/students/<student_id> con una petición DELETE al mandar un identificador de estudiante válido, o 404 si no existe ese identificador.

Parameters **`client`** (*`Flask.test_client`*) – Mock de la API para mandar peticiones durante los tests.

`test_api.`**`test_get_student`**(*client*, *valid_student*)

Testea si se recibe código 200 en la ruta /api/v1/students/<student_id> con una petición GET al mandar un identificador de estudiante válido a la vez que comprueba que el JSON recibido sea adecuado, o 404 si el identificador de estudiante no es válido.

Parameters

- **`client`** (*`Flask.test_client`*) – Mock de la API para mandar peticiones durante los tests.
- **`valid_student`** (*`dict`*) – JSON con una estructura de estudiante válida.

test_api.**test_get_students**(*client*)
Testea si se recibe código 200 en la ruta /api/v1/students con una petición GET y que el resultado sea una lista vacía ya que no hay datos previamente cargados.

Parameters **client** (*Flask.test_client*) – Mock de la API para mandar peticiones durante los tests.

test_api.**test_post_student**(*client*, *valid_student*)
Testea si se recibe código 201 en la ruta /api/v1/students con una petición POST al mandar un esquema de estudiante válido o 400 si no es válido (en este caso se envía un json vacío).

Parameters

- **client** (*Flask.test_client*) – Mock de la API para mandar peticiones durante los tests.
- **valid_student** (*dict*) – JSON con una estructura de estudiante válida.

test_api.**test_put_student**(*client*, *valid_student*)
Testea si se recibe código 204 en la ruta /api/v1/students/<student_id> con una petición PUT al mandar un esquema e identificador de estudiante válido. Si no se encuentra el estudiante comprueba si devuelve 404, o 400 si el esquema no es válido (en este caso se envía un json vacío).

Parameters

- **client** (*Flask.test_client*) – Mock de la API para mandar peticiones durante los tests.
- **valid_student** (*dict*) – JSON con una estructura de estudiante válida.

test_api.**test_status**(*client*, *status*)
Testea si se recibe código 200 en la ruta /status con una petición GET y que el resultado sea un JSON con formato {'status': "OK"}

Parameters **client** (*Flask.test_client*) – Mock de la API para mandar peticiones durante los tests.

CHAPTER 6

Despliegue en un PaaS

6.1 Heroku

La primera opción de PaaS que se ha usado es Heroku debido a su simpleza de uso para ir familiarizandome con despliegues. Para configurar un despliegue desde la herramienta de construcción, se ha incluido la regla `make heroku` en el `Makefile`. Esta regla contiene las siguientes acciones:

1. `sudo snap install heroku --classic`: Instala el CLI de Heroku en Ubuntu con el gestor de paquetes **snap**.
2. `heroku login`: Abre el navegador y nos pide introducir nuestros credenciales de Heroku para loguearnos en el CLI.
3. `heroku create notas-iv --buildpack heroku/python`: Crea nuestra aplicación con nombre **notas-iv** con el buildpack de python que ofrece Heroku (aunque no era realmente necesario incluirlo ya que Heroku detecta el lenguaje de la app automaticamente), enlazando un repostiorio remoto a nuestro repositorio local.
4. `git push heroku master`: Desplegamos nuestra aplicación en el repositorio remoto creado anteriormente.

Para saber cómo debe Heroku ejecutar nuestra aplicación, es necesario incluir un archivo llamado `Procfile` en la raiz del proyecto con el siguiente contenido:

```
web: make start-no-pm2
```

Debemos de poner el keyword `web` para decirle a Heroku que nuestra aplicación es un servicio web que recibirá peticiones HTTP y nos habilita un puerto que deberemos asignar a nuestra aplicación con la variable de entorno `$PORT`. Se ha creado una regla `start-no-pm2` ya que no tiene sentido ejecutar nuestra aplicación en pm2 si ya el propio Heroku nos proporciona las herramientas necesarias, como la escalabilidad por ejemplo.

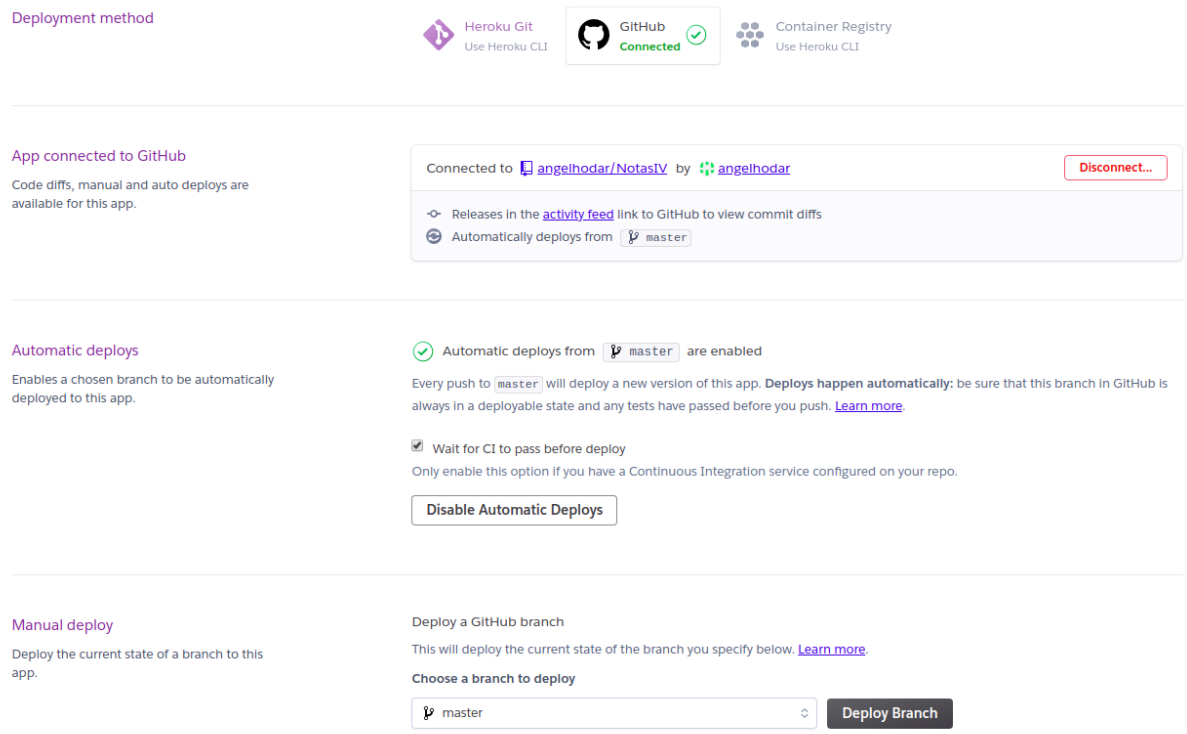
6.1.1 Heroku y GitHub

Cuando ya tenemos la aplicación desplegada, cada vez que hagamos un push a nuestro repo debemos de ejecutar `git push heroku master` para notificar a Heroku de los cambios y que actualice nuestra aplicación. Para evitar esto, podemos configurar el repo de GitHub de nuestra aplicación para que cada vez que hagamos un push a nuestro repo, automaticamente Heroku actualice la aplicación.

Para ello simplemente debemos irnos al apartado **Deploy** en la página de nuestra aplicación en Heroku y en el apartado *Deployment method* le damos a GitHub e introducimos nuestros credenciales de Github. Ahora solo faltaría seleccionar el repo correspondiente a la aplicación.

Esto crea un hook en nuestro repo que se enlazará con Heroku cada vez que se introduzca un cambio. También nos da la posibilidad de configurar si queremos que ese nuevo despliegue solo se lleve a cabo si la nueva versión de nuestro repo pasa los tests en el CI que tengamos configurado en caso de tener alguno, sin necesidad de configurar nada mas.

En la siguiente imagen se puede ver como queda todo configurado despues de terminar el proceso:



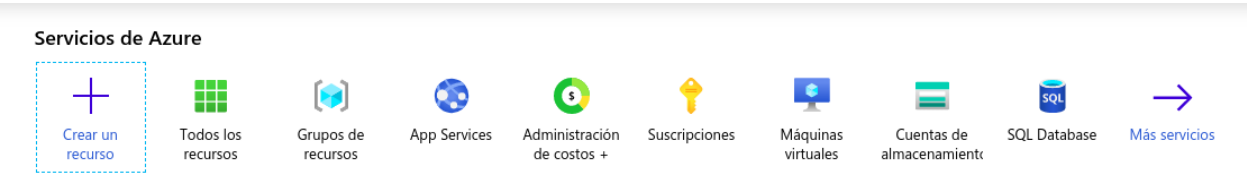
Ahora simplemente cada vez que incluyamos un cambio en nuestro repo, Heroku ya se encargará de actualizar nuestra app (siempre y cuando pase los tests correspondientes).

6.2 Azure Web Apps

Como segunda opción para complementar el despliegue con Heroku, se ha elegido Azure Web Apps ya que no pide tarjeta de crédito y gracias al programa *GitHub Student Developer Pack* dan 100$ de crédito enlazando el correo de la UGR a la cuenta.

Attention: Antes de empezar a describir cómo se ha desplegado, hay que decir que me ha resultado imposible hacerlo con el CLI que proporciona (az), ya que incluso escribiendo los comandos tal cual explican en la documentación para desplegar la aplicación, da un error de Python del propio código del CLI y no he encontrado ninguna solución al fallo, por lo que he tenido que hacerlo mediante el portal web de Azure. De igual forma, para cada paso que haga incluiré los comandos que habría que ejecutar en el CLI para realizar lo mismo que haré mediante la web.

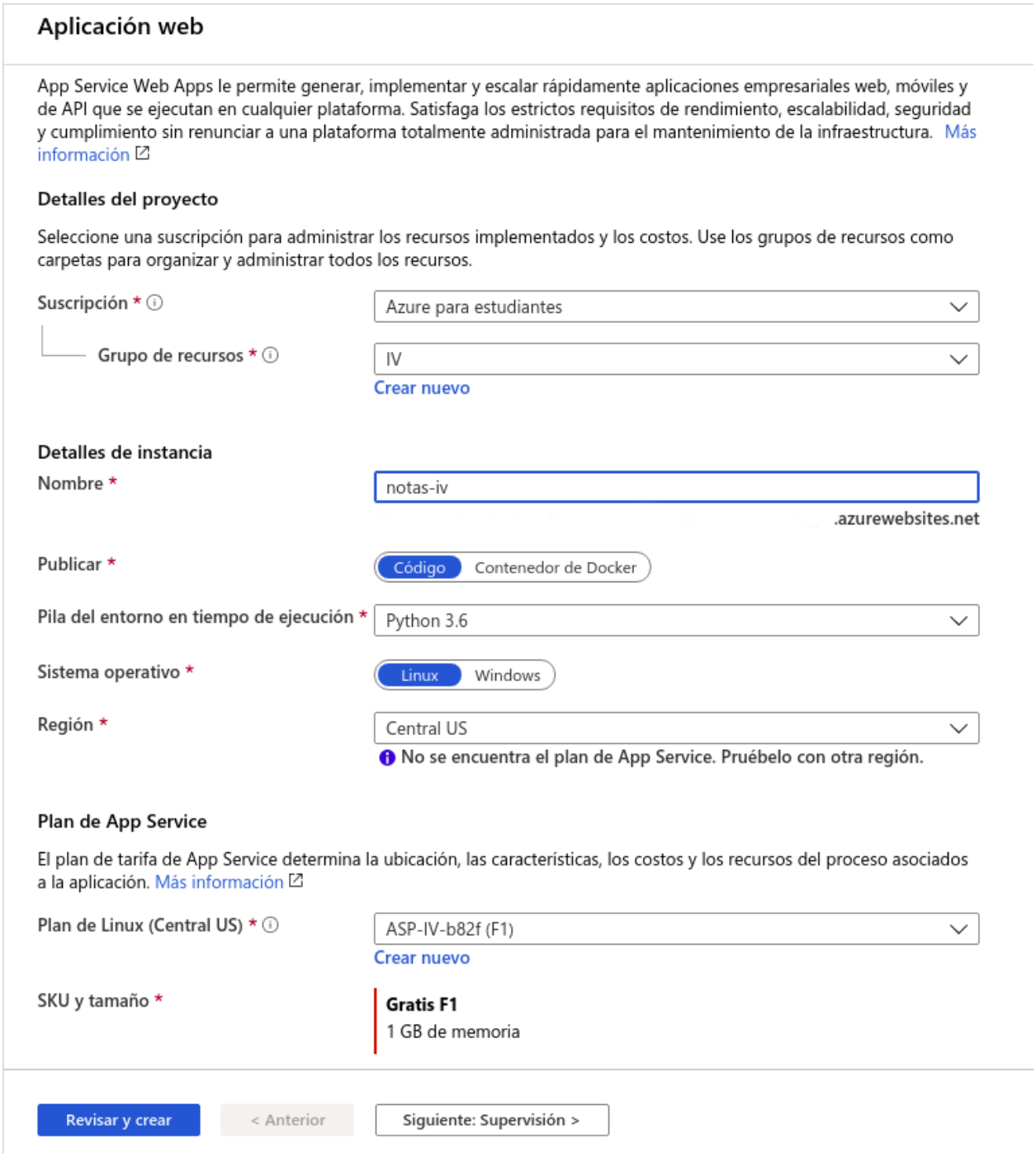
Dicho lo anterior, vamos a empezar. Una vez nos logueamos en el portal de Azure, nos saldrán unas cuantas opciones sobre lo que podemos crear en la plataforma:



Para crear una aplicación web, debemos seleccionar la opción **App Services** y nos saldrá una plantilla con varias opciones que deberemos rellenar:

- `Grupo de recursos`: Contenedor que almacena los recursos relacionados con una solución de Azure. Simplemente creamos uno nuevo dándole el nombre que queramos, en mi caso **IV**.
- `Nombre`: El nombree que queremos que tenga nuestra aplicación y que fomará parte de la URL de la misma en Azure.
- `Publicar`: En nuestro caso, como queremos desplegar mediante un push desde GitHub, seleccionaremos la opción **Código**.
- `Pila del entorno en tiempo de ejecución`: Lenguaje y versión del mismo que tendrá el contenedor de nuestra app.
- `Región`: Al usar la suscripción de Azure para estudiantes con el correo de la UGR, solo podemos seleccionar *Central US* como región.

El resto de parámetros se rellenan automaticamente. En mi caso, se rellenaría tal que asi:



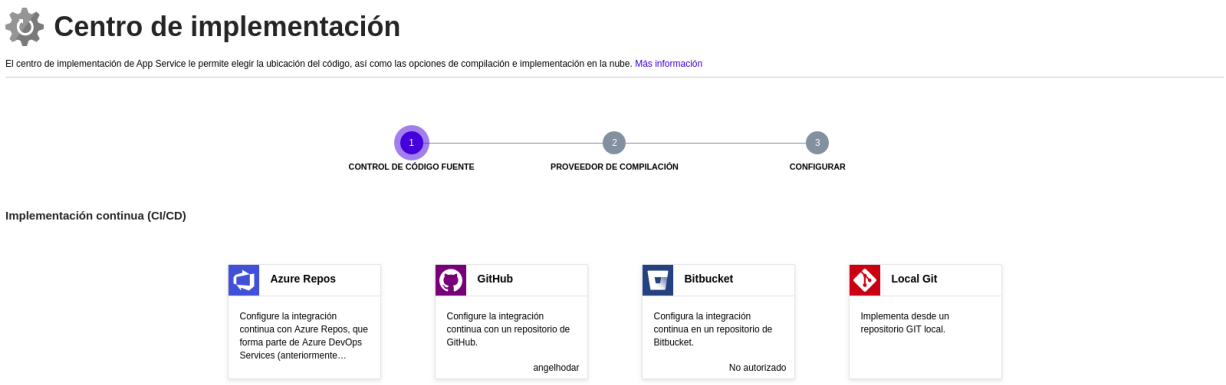
Para realizar estos mismos pasos desde el CLI, bastaría con ejecutar lo siguiente:

```
$ az login
$ az webapp up --sku F1 -n notas-iv -l centralus
```

Cuando lo tengamos todo, simplemente le damos al botón **Revisar y crear** en la parte inferior y esperamos a que se despliegue la aplicación. Una vez terminado, si accedemos a la URL de nuestra app veremos que sale una página de Azure por defecto, ya que no hemos desplegado todavía nuestro código y por defecto Azure tiene distintas plantillas en función del lenguaje de programación que le hayamos indicado.

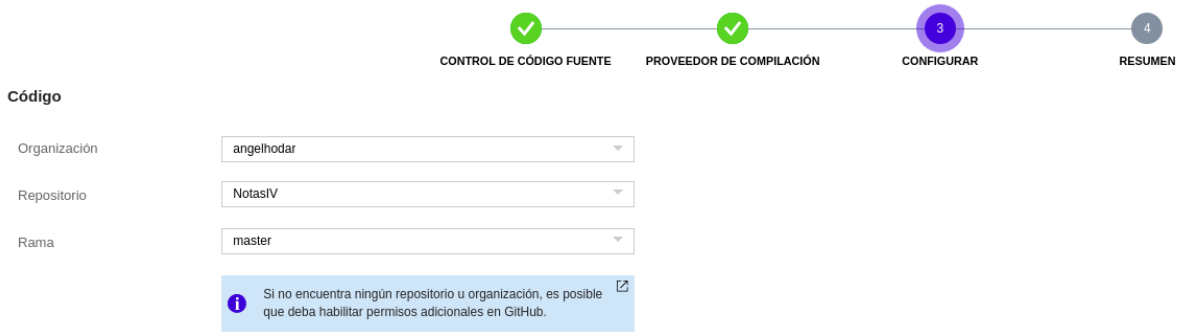
6.2.1 Azure y GitHub

Ahora es el paso de indicarle a Azure qué código queremos desplegar. Para ello nos vamos a la sección **Centro de Implementación** en la parte izquierda y veremos que nos oferce distintas posibilidades de despliegue de nuestro código:



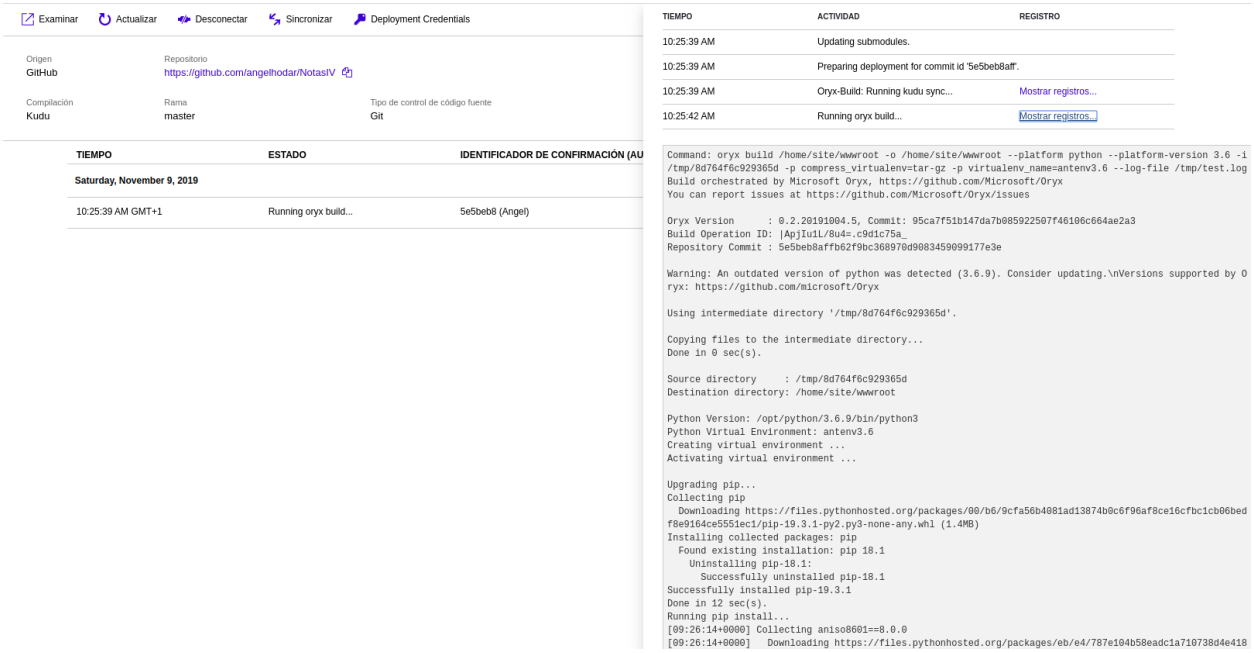
Seleccionamos **GitHub**, luego nos saldrán 2 opciones de compilación de nuestra app, en nuestro caso elegimos **Kudu**, la primera opción:

Ahora nos pedirá que introduzcamos nuestro nick de GitHub, el repo y la rama que contiene el código que queremos desplegar:



Note: Si por algun casual no salen los datos, es posible que primero haya que irse a la configuración de nuestra cuenta de GitHub y dar permisos a la aplicación Azure Web Apps en el apartado de **Aplicaciones**.

Una vez hecho esto, simplemente deberemos darle a **Finalizar** y nos saldrá una nueva tarea de compilación de la app, que montará un entorno virtual e instalará las dependencias incluidas en el archivo **requirements.txt** (se ha debido de crear tal archivo ya que no aceptan un Pipfile):



Posteriormente, Azure buscará en el directorio raiz de nuestro repositorio un archivo llamado application.py o app.py si estamos usando el microframework Flask, lo cual es nuestro caso, y lanzará un servidor de HTTP WSGI de Gunicorn en el puerto 5000:

```
# If application.py
$ gunicorn --bind=0.0.0.0 --timeout 600 application:app
# If app.py
$ gunicorn --bind=0.0.0.0 --timeout 600 app:app
```

En nuestro caso nos vale perfectamente, pero en mi caso particular estoy usando uWSGI, otra libreria parecida a Gunicorn asi que me interesaría usar ésta, aparte de indicarle cómo quiero que se ejecute mi aplicación. Para ello, Azure ofrece la posibilidad de indicar un comando de inicio en la sección de **Configuración**, donde además podemos

indicar variables de entorno (aunque en este caso no nos haria falta, he declarado una variable `$PORT`).

Esto estaría muy bien si no fuera porque no se pueden instalar paquetes adicionales como **make** para poder ejecutar **make start**, o si simplemente se le indica como comando de inicio lo siguiente:

```
$ uwsgi --http 0.0.0.0:$(PORT) --module app:app --master
```

No funciona, directamente dice que no encuentra el paquete uwsgi. Me he metido por SSH en los logs del contenedor y me he asegurado de que activa el entorno virtual antes de lanzar el comando de inicio, por lo que ese no es el problema. Aparte, se ve perfectamente en los logs que lo instaló junto con el resto de paquetes en `requirements.txt`:

```
[10:05:41+0000]     Running setup.py install for uwsgi: finished with status 'done'
```

Por lo tanto, he dejado la ejecución por defecto con Gunicorn que si funciona.

CHAPTER 7

Docker

En esta sección se documentará la creación de un contenedor de docker para correr nuestra aplicación de forma completamente aislada y solo con las dependencias estrictamente necesarias. Posteriormente, se creará un repositorio en Docker Hub con la imagen creada y se usará para desplegar en Heroku y Azure.

7.1 Creación de la imagen

Para crear la imagen necesitamos crear 2 archivos:

- `Dockerfile`: Contiene los comandos que se usarán para crear la imagen.
- `.dockerignore`: Funciona como un .gitignore, nos permite indicar qué archivos queremos que no se añadan a nuestra imagen ya que son innecesarios (por ejemplo el directorio **.git**).

El archivo Dockerfile está documentado paso por paso y contiene lo siguiente:

```
# Usamos la versión alpine de la versión 3.7 de python yaa que es
# mucho mas ligera (100MB vs 1GB)
FROM python:3.7-alpine

# Datos propios
LABEL maintainer="Ángel Hódar (angelhodar76@gmail.com)"

# Exponemos el puerto de la variable de entorno
EXPOSE $PORT

# Copiamos primero solo el requirements para aprovecharnos del sistema
# de layers de las imagenes docker e instalamos las dependencias
COPY requirements.txt /tmp
RUN cd /tmp && pip install -r requirements.txt

# Copiamos los archivos (solo los no añadidos en el .dockerignore)
COPY . /app
# Nos movemos al directorio creado previamente.
```

(continues on next page)

(continued from previous page)

```
WORKDIR /app

# Finalmente ejecutamos la app escuchando en el puerto definido en PORT
CMD gunicorn -b 0.0.0.0:${PORT} app:app
```

Una vez tenemos el `Dockerfile` creado, debemos situarnos en el mismo directorio y ejecutar:

```
$ docker build -t notas-iv .

Sending build context to Docker daemon  97.28kB
Step 1/7 : FROM python:3.7-alpine
---> 8922d588eec6
Step 2/7 : EXPOSE $PORT
---> Using cache
---> 82303a8b75d2
Step 3/7 : COPY requirements.txt /tmp
---> Using cache
---> 2027a0d77ead
Step 4/7 : RUN cd /tmp && pip install -r requirements.txt
---> Using cache
---> 90833d40b7ba
Step 5/7 : COPY . /app
---> Using cache
---> 17725ae98b9b
Step 6/7 : WORKDIR /app
---> Using cache
---> dc00b49afebb
Step 7/7 : CMD gunicorn -b 0.0.0.0:${PORT} app:app
---> Using cache
---> 2a90f78a4ae6
Successfully built 2a90f78a4ae6
Successfully tagged notas-iv:latest
```

Esto creará una imagen llamada **notas-iv**, indicando con **.** el path donde está nuestro Dockerfile. En este caso como ya se ha ejecutado previamente, vemos como usa la caché para no tener que ejecutar cada comando de nuevo. Esto es especialmente interesante en el caso de la instalación de dependencias con **pip**, ya que solo se ejecutará si cambiamos alguna dependencia, en lugar de hacerse siempre que hagamos un cambio en el código por ejemplo.

Para listar las imagenes que tenemos creadas podemos ejecutar:

```
$ docker images

REPOSITORY            TAG                 IMAGE ID            CREATED             SIZE
notas-iv              latest              0287ab292cdb        20 minutes ago      ␣
↪126MB
```

Para comprobar que la imagen funciona como debe, simplemente debemos arrancar un contenedor de esa imagen. Para ello, simplemente ejecutamos:

```
$ docker run -e PORT=$PORT -p $HOST_PORT:$PORT notas-iv

[2019-11-21 15:04:40 +0000] [1] [INFO] Starting gunicorn 19.9.0
[2019-11-21 15:04:40 +0000] [1] [INFO] Listening at: http://0.0.0.0:5000 (1)
[2019-11-21 15:04:40 +0000] [1] [INFO] Using worker: sync
[2019-11-21 15:04:40 +0000] [7] [INFO] Booting worker with pid: 7
```

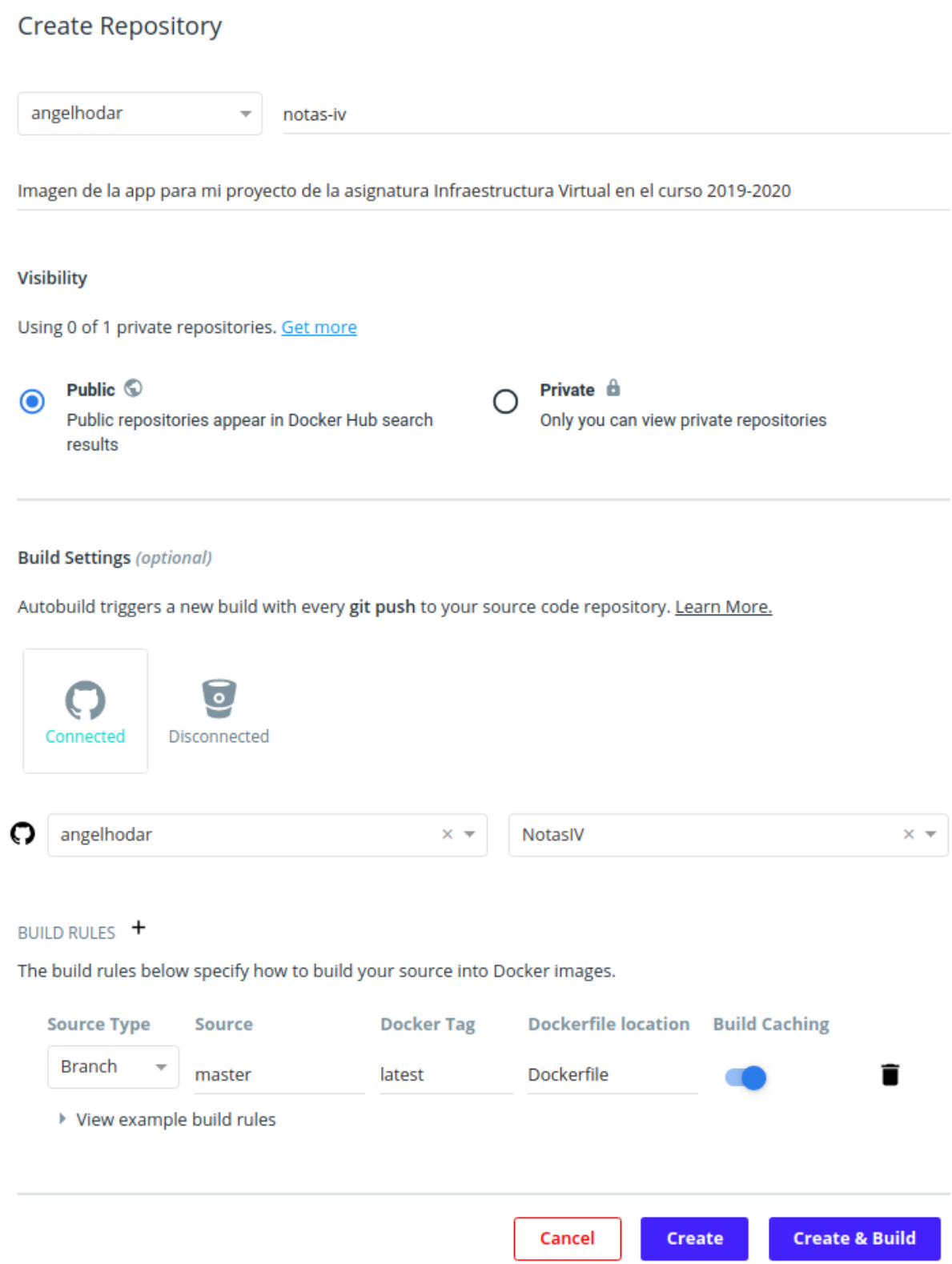
La opción `-p` le indica que vamos a mapear el puerto `$HOST_PORT` del anfitrión al puerto `$PORT` del contenedor.

Basicamente esto quiere decir que si queremos acceder a nuestro contenedor localmente en la dirección **127.0.0.1:550** por ejemplo, el valor de `$HOST_PORT` debe de ser **550**, mientras que el valor de `$PORT` deberá de ser aquel en el que queremos que escuche nuestra app web dentro de su contenedor (en el caso de la ejecución anterior era **5000**).

Además, con la opción `-e` hacemos que el servidor WSGI de Gunicorn se ejecute escuchando en el puerto definido en la variable de entorno `$PORT` mencionada anteriormente, algo necesario también posteriormente cuando se ejecuta en Heroku.

7.2 Docker Hub

Ahora que ya tenemos nuestra imagen creada y funcionando, vamos a desplegarla en Docker Hub. Para ello primero nos registramos, y cuando lo hayamos hecho le damos al boton **Create Repository** en en apartado de **Repositories**. Para automatizar la actualizacion de la imagen cada vez que hagamos un push a nuestro repositorio, Docker Hub nos da directamente la opción de enganchar un repositorio de GitHub desde donde obtener los datos para construir la imagen.



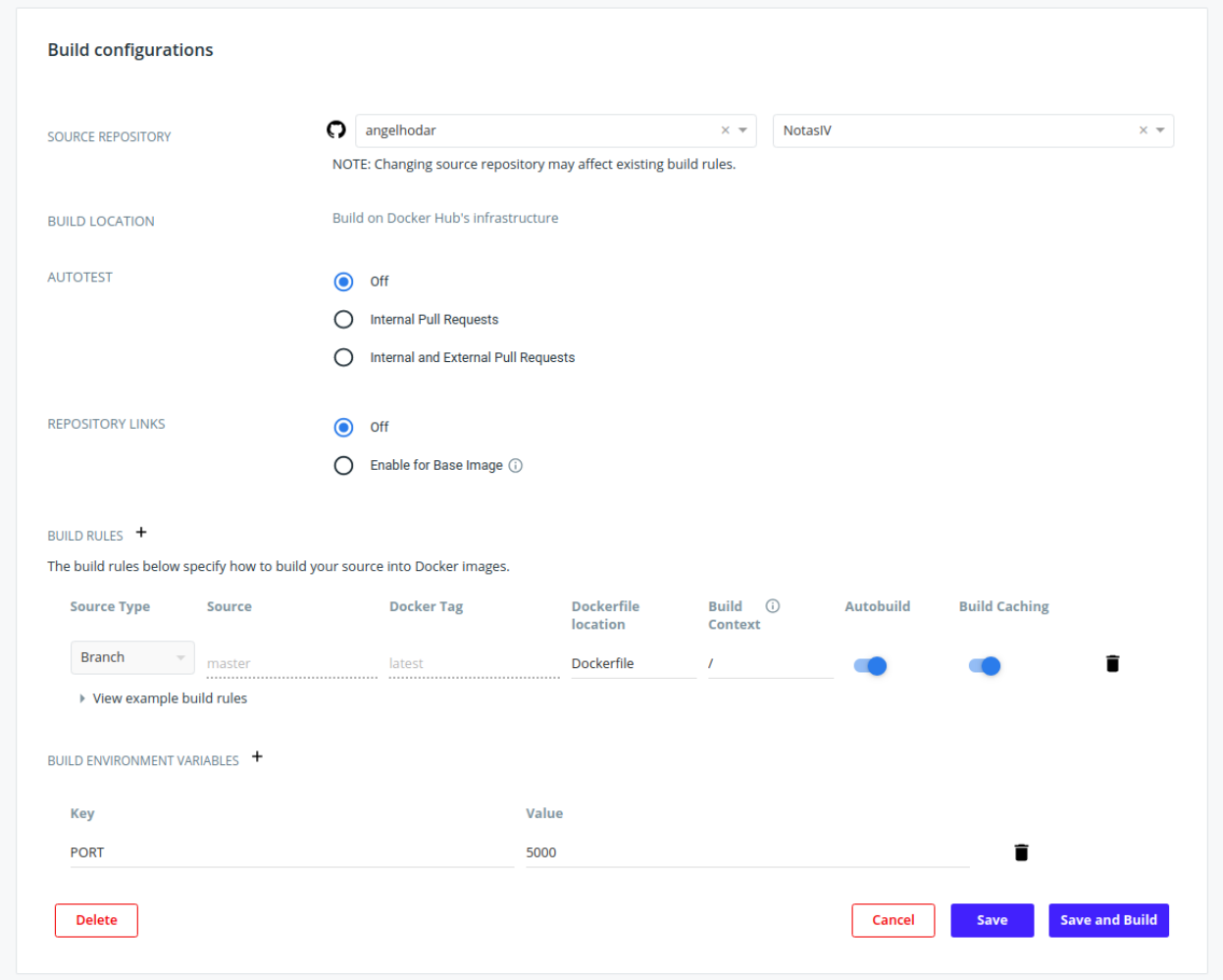
Si queremos que Docker Hub obtenga la información necesaria desde el repositorio de GitHub que le hemos asignado, deberemos darle a **Create & Build**. Si por el contrario queremos subir la imagen manualmente, le damos a **Create**.

Si elegimos la segunda opción, debemos ejecutar tan solo 3 comandos para subir la imagen a Docker Hub:

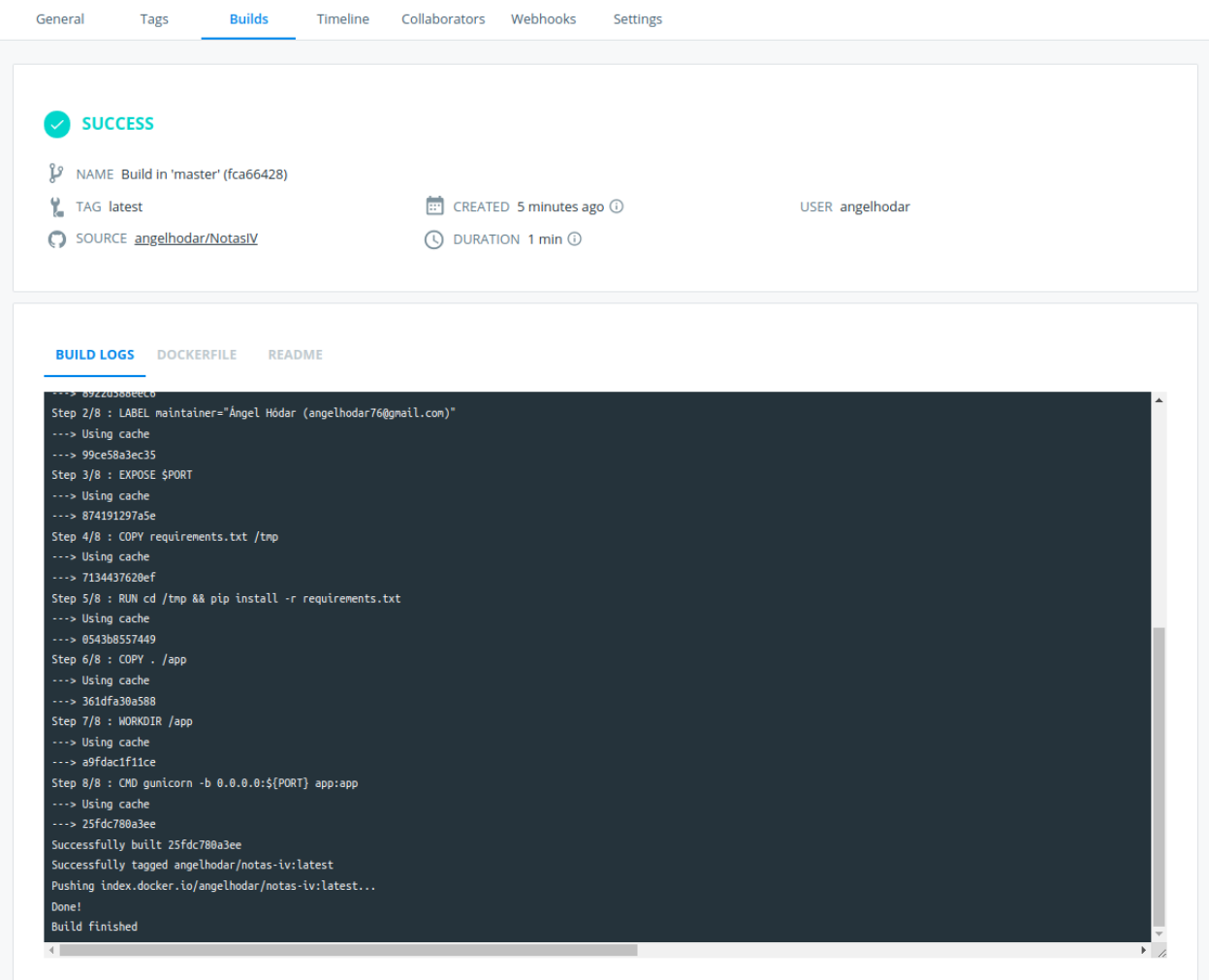
```
# Nos logueamos a nuestra cuenta de Docker Hub
$ docker login

# Cambiamos el nombre de la imagen con el del repo, añadiendo el tag que queramos.
$ docker tag notas-iv angelhodar/notas-iv:latest

# Sube la imagen al repo remoto.
$ docker push angelhodar/notas-iv:latest
```

Ahora necesitamos un paso extra para automatizar las builds con GitHub, ya que si seleccionamos **Create & Build** solo se crea en ese momento, pero no se tienen en cuenta futuros `git push` que se hagan en el repo.

En nuestro repositorio de Docker Hub, debemos irnos al apartado Builds y configurarlo de esta manera:



Una vez lo hayamos hecho, le damos al botón **Save and Build** y se iniciará una nueva build de nuestra imagen. Cuando se complete nos deberá salir un resultado tal que asi:



Y con esto ya tendriamos Docker Hub completamente configurado y automatizado con nuestro repo de GitHub.

7.3 Depligue en Heroku

Desplegar nuestra imagen en Heroku es igual de sencillo que anteriormente. Para empezar, necesitamos estar logueados en Heroku desde el CLI, asi que ejecutamos:

```
$ heroku login
```

Seguimos los pasos que nos indica y ahora creamos nuestra app con el nombre que queramos, en mi caso va a ser **notas-iv**.

```
$ heroku create notas-iv

Creating  notas-iv... done
https://notas-iv.herokuapp.com/ | https://git.heroku.com/notas-iv.git
```

Una vez tenemos la app creada, necesitamos crear un archivo llamado `heroku.yml`, que tendrá una funcionalidad parecida al `Procfile`, pero esta vez se encargará de montar y correr la imagen que definamos en el Dockerfile en la app que hemos creado previamente. El archivo tiene el siguiente formato:

```
build:
    docker:
        web: Dockerfile
```

Si nos fijamos es muy similar al `Procfile`, tan solo le estamos diciendo que para montar nuestra app se va a usar docker utilizando el proceso `web`, y que todas las reglas necesarias para hacerlo están en nuestro Dockerfile. Cuando lo tengamos listo simplemente debemos cambiar el stack de nuestra app a modo contenedor con el siguiente comando:

```
$ heroku stack:set container

Stack set. Next release on  notas-iv will use container.
Run git push heroku master to create a new release on  notas-iv.
```

Esto provoca que heroku busque el archivo `heroku.yml` en lugar del `Procfile`. Y ya solo falta hacer push de nuestra app a Heroku (pongo algunas lineas de la salida para que se vea como usa el Dockerfile en lugar del Procfile):

```
$ git push heroku master

Contando objetos: 437, listo.
Delta compression using up to 8 threads.
Comprimiendo objetos: 100% (419/419), listo.
Escribiendo objetos: 100% (437/437), 1.03 MiB | 11.93 MiB/s, listo.
Total 437 (delta 257), reused 0 (delta 0)
remote: Compressing source files... done.
remote: Building source:
remote: === Fetching app code
remote:
remote: === Building web (Dockerfile)
remote: Sending build context to Docker daemon   98.3kB
remote: Step 1/8 : FROM python:3.7-alpine
remote: 3.7-alpine: Pulling from library/python
remote: 89d9c30c1d48: Pulling fs layer
remote: 910c49c00810: Pulling fs layer
remote: 7efe415eb85a: Pulling fs layer
remote: 7d8d53519b81: Pulling fs layer
remote: 519124ac136c: Pulling fs layer
remote: 7d8d53519b81: Waiting
remote: 519124ac136c: Waiting
remote: 910c49c00810: Download complete
remote: 7d8d53519b81: Verifying Checksum
remote: 7d8d53519b81: Download complete
remote: 519124ac136c: Download complete
remote: 89d9c30c1d48: Verifying Checksum
remote: 89d9c30c1d48: Download complete
remote: 7efe415eb85a: Verifying Checksum
remote: 7efe415eb85a: Download complete
remote: 89d9c30c1d48: Pull complete
remote: 910c49c00810: Pull complete
remote: 7efe415eb85a: Pull complete
remote: 7d8d53519b81: Pull complete
remote: 519124ac136c: Pull complete
remote: Digest:␣
↪sha256:de9fc5bc46cb1a7e2222b976394ea8aa0290592e3075457d41c5f246f176b1bf
remote: Status: Downloaded newer image for python:3.7-alpine
remote:  ---> 8922d588eec6
remote: Step 2/8 : LABEL maintainer="Ángel Hódar (angelhodar76@gmail.com)"
remote:  ---> Running in 41e15861418b
```

(continues on next page)

(continued from previous page)

```
remote: Removing intermediate container 41e15861418b
remote:  ---> c3e60445ce92
remote: Step 3/8 : EXPOSE $PORT
remote:  ---> Running in d619d3dc2b1b
remote: Removing intermediate container d619d3dc2b1b
remote:  ---> 804c40f0f5d6
remote: Step 4/8 : COPY requirements.txt /tmp
remote:  ---> d6fe1b20c339
remote: Step 5/8 : RUN cd /tmp && pip install -r requirements.txt
remote:  ---> Running in 9cfb5b9f4ed2
```

Una salida importante de este comando que no he mostrado es la siguiente:

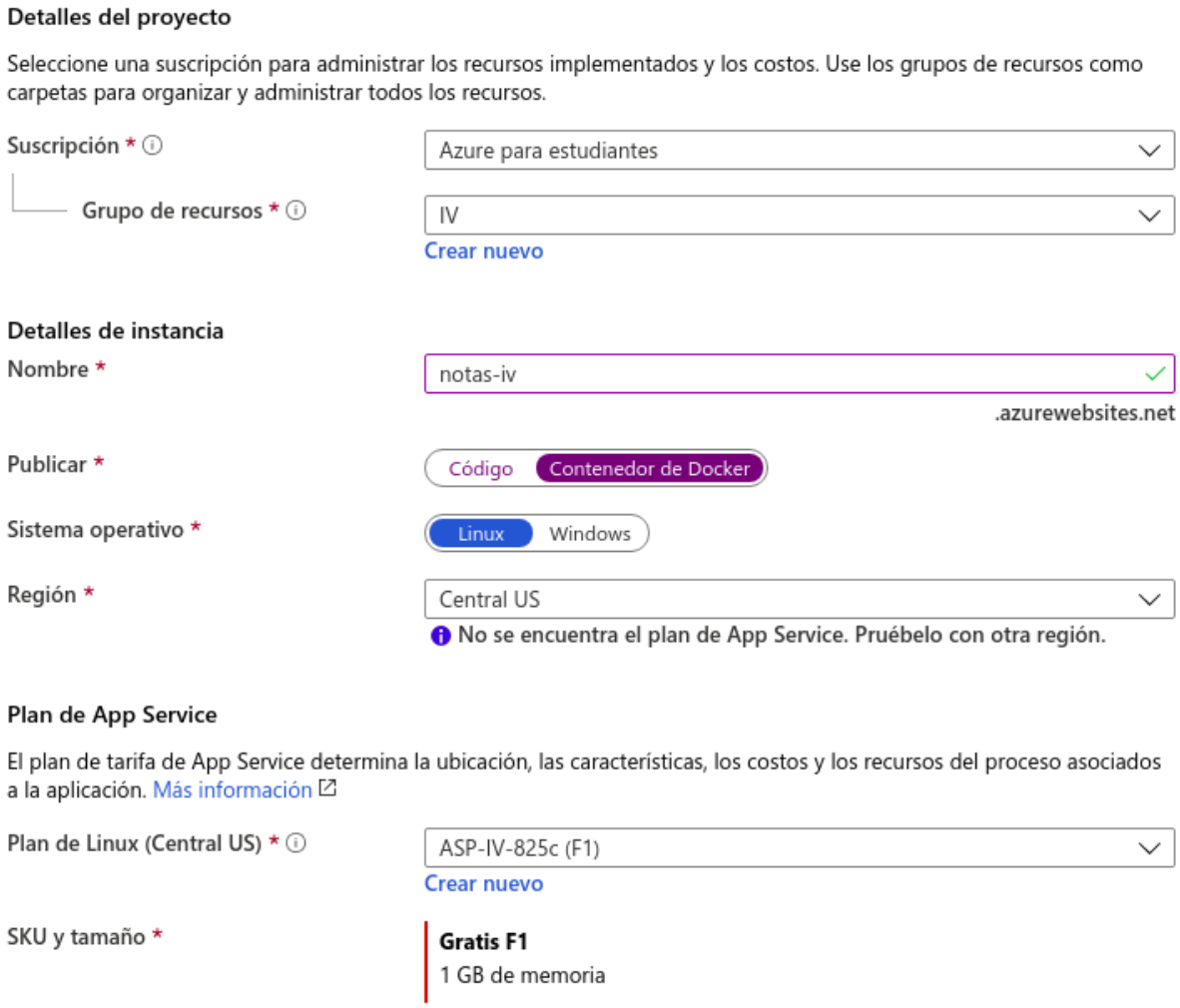
```
remote: === Pushing web (Dockerfile)
remote: Tagged image "6fb25269ffa2f2f648ce1fbeaf7de9a083b67b18" as "registry.heroku.
↪com/notas-iv/web"
remote: The push refers to repository [registry.heroku.com/notas-iv/web]
```

Basicamente lo que se hace cuando ejecutamos el comando anterior es hacer un pull de la imagen al **Container Registry** de Heroku poniendole ese nombre concreto a la imagen (registry.heroku.com/notas-iv/web), que contiene el nombre de nuestra app y el tipo de proceso que requiere nuestra aplicación (web), igual que cuando se definía en el `Procfile`.

Ya solo faltaría asociar la app al repo de GitHub para que se ejecute el despliegue con tan solo hacer git push a nuestro repo, tal y como se hizo en la sección *Heroku y GitHub*.

7.4 Despliegue en Azure

Desplegar en Azure es tremendamente sencillo, tan solo debemos crear un nuevo App Service y especificarle que queremos usar un contenedor Docker:



Al seleccionar docker, se nos abrirá una nueva pestaña en la que deberemos indicar qué imagen queremos usar y de dónde extraerla. En nuestro caso, el proveedor de imagenes sería Docker Hub, especificandole la ruta completa de nuestra imagen, con el nombre del repo y el tag que queremos usar:

Ahora tan solo debemos darle a **Revisar y crear** y nuestro contenedor estará desplegado y funcionando. Resulta extraño que no tengamos que configurar ningun parámetro adicional, como la variable de entorno `$PORT` que necesita el contenedor para funcionar. Pero, si nos fijamos en los logs que nos proporciona Azure, podemos ver cómo han ejecutado la imagen:

```
$ docker run -d -p 7530:80 --name notas-iv_0_b168528e -e PORT=80 -e WEBSITE_SITE_
↪NAME=notas-iv -e WEBSITE_HOSTNAME=notas-iv.azurewebsites.net
```

Hay incluso más variables de entorno en el comando, pero la que nos interesa especialmente es que utilizan una variable `$PORT`. En concreto, tiene valor **80**, algo esperable al tratarse de una app web.

CHAPTER 8

Creación y aprovisionamiento

En esta sección vamos a crear una máquina virtual, que aprovisionaremos con todo lo necesario para poder ejecutar nuestra app.

8.1 Creación de la VM

Para crear la VM vamos a usar **Vagrant**, que nos permitirá tener un archivo de configuración para establecer la box que vamos a usar y el proveedor que ejecutará la VM (en este caso he elegido **VirtualBox** por ser gratis y por su portabilidad), además de otros ajustes. También podemos especificarle directamente el sistema de aprovisionamiento que vamos a usar, que en mi caso ha sido **ansible**, asi podemos tener todo lo necesario con el comando vagrant.

Para configurarlo todo, tan solo necesitamos crear un archivo con nombre `Vagrantfile`, que tiene el siguiente formato:

```
# -*- mode: ruby -*-
# vi: set ft=ruby :

Vagrant.configure("2") do |config|
    # Definimos el nombre de nuestra VM para Vagrant
    config.vm.define "NotasIV"
    # He elegido Ubuntu 18.04 LTS ya que es la última version
    # estable y con mayor tiempo de soporte actualmente de Ubuntu
    config.vm.box = "ubuntu/bionic64"
    # Con esto evitamos que busque actualizaciones de la box automaticamente
    # Mejor actualizar manualmente que en un posible descuido
    config.vm.box_check_update = false
    # Asociamos el acceso a la VM a través de 127.0.0.1, asociando el puerto 5000
    # del anfitrion al puerto 5000 de la VM.
    config.vm.network "forwarded_port", guest: 5000, host: 5000, host_ip: "127.0.0.1"

    # Localmente he usado virtualbox
    config.vm.provider "virtualbox" do |vb|
        # Configuramos el nombre que queremos que tenga la VM dentro de virtualbox
```

(continues on next page)

(continued from previous page)

```
        # para que no nos ponga nombres raros vagrant
        vb.name = "NotasIV"
        # Aparte le definimos 1GB de RAM y 2 nucleos de CPU
        vb.memory = "1024"
        vb.cpus = 2
    end

    # Simplemente configuramos ansible y la ruta de nuestro playbook
    # para el aprovisionamiento
    config.vm.provision "ansible" do |ansible|
        ansible.playbook = "provisioning/playbook.yml"
    end
end
```

Note: Como SO base he seleccionado la última versión estable de Ubuntu, la 18.04 LTS, ya que no es muy pesada (307MB) y tiene las últimas actualizaciones de la distribución.

Una vez tenemos este archivo, con la herramienta de construcción simplemente ejecutamos:

```
$ make vm
```

Esto lo que hará será crearnos una máquina virtual con los ajustes que hayamos definido en el `Vagrantfile`, pero **no** aprovisionará la máquina. En concreto, devolverá el siguiente output:

```
    Bringing machine 'NotasIV' up with 'virtualbox' provider...
==> NotasIV: Importing base box 'ubuntu/bionic64'...
==> NotasIV: Matching MAC address for NAT networking...
==> NotasIV: Setting the name of the VM: NotasIV
==> NotasIV: Clearing any previously set network interfaces...
==> NotasIV: Preparing network interfaces based on configuration...
    NotasIV: Adapter 1: nat
==> NotasIV: Forwarding ports...
    NotasIV: 5000 (guest) => 5000 (host) (adapter 1)
    NotasIV: 22 (guest) => 2222 (host) (adapter 1)
==> NotasIV: Running 'pre-boot' VM customizations...
==> NotasIV: Booting VM...
==> NotasIV: Waiting for machine to boot. This may take a few minutes...
    NotasIV: SSH address: 127.0.0.1:2222
    NotasIV: SSH username: vagrant
    NotasIV: SSH auth method: private key
    NotasIV: Warning: Remote connection disconnect. Retrying...
    NotasIV:
    NotasIV: Vagrant insecure key detected. Vagrant will automatically replace
    NotasIV: this with a newly generated keypair for better security.
    NotasIV:
    NotasIV: Inserting generated public key within guest...
    NotasIV: Removing insecure key from the guest if it's present...
    NotasIV: Key inserted! Disconnecting and reconnecting using new SSH key...
==> NotasIV: Machine booted and ready!
==> NotasIV: Checking for guest additions in VM...
    NotasIV: Guest Additions Version: 5.2.34
    NotasIV: VirtualBox Version: 6.0
==> NotasIV: Mounting shared folders...
    NotasIV: /vagrant => /home/angel/GitHub/NotasIV
```

Para acceder a ella, podemos hacerlo con el siguiente comando:

```
$ vagrant ssh
```

Esto funciona porque cuando vagrant crea nuestra máquina, también crea un usuario llamado `vagrant`, generando un par de llaves SSH e insertando la pública en la máquina virtual y la privada en la ruta `.vagrant/machines/NotasIV/virtualbox/private_key`, que es de donde la obtiene a la hora de hacer ssh. De hecho el proceso de creación del par de llaves y la inserción de la pública se muestra en parte de la salida de cuando levantamos la máquina:

```
NotasIV: Vagrant insecure key detected. Vagrant will automatically replace
NotasIV: this with a newly generated keypair for better security.
NotasIV:
NotasIV: Inserting generated public key within guest...
NotasIV: Removing insecure key from the guest if it's present...
NotasIV: Key inserted! Disconnecting and reconnecting using new SSH key...
```

Esto lo vamos a modificar en el aprovisionamiento, creando un usuario dentro de la máquina y asociandole el par de llaves que nosotros queramos.

8.2 Aprovisionamiento

Como se ha dicho anteriormente, para aprovisionar la máquina se ha usado ansible, y para decirle qué debe aprovisionar sobre la máquina concretamente he creado un archivo `playbook.yml` en el directorio `provisioning`, que contiene lo siguiente:

```
---
# Como Vagrant nos crea un inventario, aqui podemos poner directamente el nombre que␣
↪le dimos a la VM.
- hosts: NotasIV
  tasks:
    # Primero con apt vamos a varias dependencias, como pip, make y npm para usar pm2
    - name: Instalar dependencias
    become: true
    apt:
        name:
        - git
        - python3-pip
        - nodejs
        - npm
        - make
        state: present
        # Esto ejecuta sudo apt update antes de instalar las dependencias, necesario
        # para que encuentre el paquete python3-pip
        update_cache: true

    # Una vez tenemos npm ahora instalamos pm2 de forma global en el equipo para que
    # cualquier usuario que creemos tenga acceso.
    - name: Instalar pm2 globalmente
    become: true
    npm:
        name: pm2
        global: yes

    # Instalamos pipenv para tener las dependencias del proyecto aisladas del resto
```

(continues on next page)

(continued from previous page)

```
    # de la VM
    - name: Instalar pipenv
    pip:
        name: pipenv

    # Me creo un usuario angel con una shell de bash. Por defecto le crea un home, no␣
↪hace
    # falta especificarselo
    - name: Crear usuario angel
    become: true
    user:
        name: angel
        shell: /bin/bash

    # Como queremos configurar este usuario por ssh para acceder a él desde el␣
↪anfitrion,
    # le mandamos la clave pública que queremos tener autorizada para ese usuario,
    # especificandole la tura en el anfitrion
    - name: Agregar clave publica para el usuario angel
    become: true
    authorized_key:
        user: angel
        state: present
        key: "{{ lookup('file', '/home/angel/.ssh/id_rsa.pub') }}"
```

Note: Un detalle importante es que, como explico en el propio playbook al principio, Vagrant nos crea un inventario para ansible en `.vagrant/provisioners/ansible/inventory/vagrant_ansible_inventory` con las maquinas que hayamos definido en el `Vagrantfile`. Como se definió una máquina de nombre NotasIV, podemos ponerla directamente en la clave hosts. Si tuvieramos mas máquinas podriamos agruparlas en un grupo y especificar ese grupo, o simplemente usar el keyword **all** para ejecutar las tasks del playbook sobre todas las maquinas definidas en el Vagrantfile. Si estuvieramos usando el comando **ansible-playbook** en lugar de vagrant, el inventario por defecto estaría en `/etc/ansible/hosts`.

Una vez tenemos todo listo para aprovisionar la máquina, ejecutamos lo siguiente:

```
$ make provision
```

Lo cual generará un output como el siguiente al ejecutarlo por primera vez:

```
NotasIV: Running ansible-playbook...

PLAY [NotasIV] *****************************************************************

TASK [Gathering Facts] *********************************************************
ok: [NotasIV]

TASK [Instalar dependencias] ***************************************************
changed: [NotasIV]

TASK [Instalar pm2 globalmente] ************************************************
changed: [NotasIV]

TASK [Instalar pipenv] *********************************************************
changed: [NotasIV]
```

(continues on next page)

(continued from previous page)

```
TASK [Crear usuario angel] ******************************************************
changed: [NotasIV]

TASK [Agregar clave publica para el usuario angel] ******************************
changed: [NotasIV]

PLAY RECAP **********************************************************************
NotasIV  : ok=6  changed=5  unreachable=0  failed=0  skipped=0  rescued=0  ignored=0
```

Las tareas marcadas con **changed** viene a decir que esa tarea se ha realizao y ha cambiado el estado de la máquina. Si por el contrario pusiera **ok**, significaría que esa tarea ya ha sido ejecutada y tenemos el sistema con el estado requerido para esa tarea, por lo que no es necesario ejecutarla.

Veamos a grandes rasgos qué hace nuestro playbook:

1. Usando el modulo **apt** de ansible, instala y actualiza las dependencias necesarias para crear el entorno necesario para ejecutar la app.
2. Usando los modulos **npm** y **pip**, instalamos pm2 y pipenv, necesarias para tener control sobre la ejecución de nuestra app y las librerías necesarias.
3. Creamos un usuario llamado *angel* con el módulo **user**, asignándole un shell de bash en lugar de sh que es el que viene por defecto.
4. Al usuario le asignamos la llave pública del par que vamos a usar para conectarnos a la máquina con ese usuario.

Para conectarnos con ssh a la máquina usando el usuario *angel* que hemos creado en el aprovisionamiento, debemos hacerlo con el comando **ssh** en lugar de **vagrant ssh**. Como vagrant asocia el puerto 2222 a ssh en la máquina y además tiene asociado *127.0.0.1* como IP de acceso, tan solo debemos ejecutar:

```
$ ssh angel@localhost -p 2222
```

Note: Suponemos que tenemos la llave privada asociada a ese usuario en `~/.ssh` en nuestro anfitrión, de lo contrario deberiamos de especificarselo al comando ssh con la opción **-i**.

8.3 Vagrant Cloud

Para subir mi box a Vagrant Cloud y que cualquiera pueda usarla simplemente nos creamos una cuenta y creamos un nuevo repositorio (realmente es muy parecido a Docker Hub). Una vez lo hayamos hecho, primero debemos exportar nuestra máquina. Para ello ejecutamos:

```
$ vagrant package --output NotasIV
```

Esto nos exportará la máquina en formato .box, y en nuestro repositorio especificaremos una versión y un proveedor, como en nuestro caso ha sido virtualbox pues lo seleccionamos y subimos la máquina.

Hint: La box se puede encontrar aquí

CHAPTER 9

Despliegue de la VM en Azure

En esta sección vamos a utilizar un servicio en la nube (en este caso Azure) para alojar la VM que crearemos con Vagrant y que aprovisionaremos usando ansible.

9.1 Configuración de Azure

Antes de empezar, necesitamos configurar algunas cosas con el CLI de Azure para que la creación de la VM se pueda llevar a cabo. En concreto, necesitamos:

- Un grupo de recursos
- Una serie de variables de entorno (como nuestra ID de suscripción a Azure).

Para crear un grupo de recursos, tan solo debemos ejecutar la siguiente orden:

```
$ az group create -l westeurope -n Hito7
```

Con esto creamos un grupo de recursos llamado **Hito7** con la opción `-n`, y también le especificamos una región que queramos con `-l`.

Note: Según la región que se le especifique, tendremos acceso a una serie de máquinas u otras, que se pueden ver desde este enlace.

Ahora solo nos faltaria obtener las variables con los credenciales necesarios, que podemos hacerlo simplemente ejecutando el siguiente comando:

```
$ az ad sp create-for-rbac
```

Que devolverá un JSON como el siguiente (se han cambiado los valores de las credenciales por **-**):

```
{
    "appId": "-----------------",
```

(continues on next page)

(continued from previous page)

```
    "displayName": "azure-cli-2019-12-23-09-47-36",
    "name": "http://azure-cli-2019-12-23-09-47-36",
    "password": "--------------",
    "tenant": "----------------"
}
```

Con esto ya tenemos todo lo necesario para empezar a configurar nuestro **Vagrantfile** en la siguiente sección.

9.2 Configuración de Vagrant

Una vez hemos obtenido en la sección anterior los credenciales necesarios, primero debemos instalar el plugin de azure para vagrant, que se encuentra aqui y que se puede llevar a cabo con el siguiente comando:

```
$ vagrant plugin install vagrant-azure
```

Ahora ya si podemos centrarnos en el **Vagrantfile**, que tiene la siguiente estructura:

```
Vagrant.configure("2") do |config|
    # Nombre de la VM para vagrant y ansible
    config.vm.define "NotasIV"

    # Necesario para el plugin de Azure
    config.vm.box = "azure"

    # Especificamos el dummy box, el cual nos proporcionará una base para nuestra␣
↪máquina.
    config.vm.box_url = 'https://github.com/msopentech/vagrant-azure/raw/master/dummy.
↪box'

    # Clave privada del par usado para conectarse a la VM
    config.ssh.private_key_path = '~/.ssh/id_rsa'

    config.vm.provider :azure do |azure, override|
        # Credenciales guardadas en variables de entorno necesarias para poder
        # desplegar (la obtención de las mismas se explica en la documentación).
        azure.tenant_id = ENV['AZURE_TENANT_ID']
        azure.client_id = ENV['AZURE_CLIENT_ID']
        azure.client_secret = ENV['AZURE_CLIENT_SECRET']
        azure.subscription_id = ENV['AZURE_SUBSCRIPTION_ID']

        # Nombre de la máquina virtual en Azure
        azure.vm_name = "notasiv"
        # El tipo de máquina, este modelo tiene 1 CPU y 1GB de RAM, aparte de ser
        # de los mas baratos
        azure.vm_size = "Standard_B1s"
        # Abrimos el puerto donde escuchará nuestra app (que lo tenemos también
        # como variable de entorno).
        azure.tcp_endpoints = ENV['PORT']
        # Especificamos la imagen que vamos a montar en nuestra máquina, en este caso␣
↪Ubuntu 18.04
        azure.vm_image_urn = 'Canonical:UbuntuServer:18.04-LTS:latest'
        # Grupo de recursos en Azure donde se creará la máquina
        azure.resource_group_name = 'Hito7'
```

(continues on next page)

(continued from previous page)

```
    end

    # Usamos ansible como servicio de provisionamiento y le especificamos la ruta
    # del playbook
    config.vm.provision "ansible" do |ansible|
        ansible.playbook = "provisioning/playbook.yml"
    end

end
```

Note: Para ver una lista de las imágenes de SO que tenemos disponibles en Azure, podemos hacerlo ejecutando `az vm image list --output table` y ver la columna **Urn** del SO que queramos, cuyo valor es lo que deberemos especificarle al parámetro az.vm_image_urn en el Vagrantfile.

Una vez tenemos este archivo, con la herramienta de construcción simplemente ejecutamos:

```
$ make vm
```

Esto lo que hará será crearnos una máquina virtual con los ajustes que hayamos definido en el `Vagrantfile`, pero **no** aprovisionará la máquina. En concreto, devolverá el siguiente output:

```
Bringing machine 'NotasIV' up with 'azure' provider...
==> NotasIV: Launching an instance with the following settings...
==> NotasIV:  -- Management Endpoint: https://management.azure.com
==> NotasIV:  -- Subscription Id: ----------------------------
==> NotasIV:  -- Resource Group Name: Hito7
==> NotasIV:  -- Location: westeurope
==> NotasIV:  -- Admin Username: vagrant
==> NotasIV:  -- VM Name: notasiv
==> NotasIV:  -- VM Storage Account Type: Premium_LRS
==> NotasIV:  -- VM Size: Standard_B1s
==> NotasIV:  -- Image URN: Canonical:UbuntuServer:18.04-LTS:latest
==> NotasIV:  -- TCP Endpoints: 5000
==> NotasIV:  -- DNS Label Prefix: notasiv
==> NotasIV:  -- Create or Update of Resource Group: Hito7
==> NotasIV:  -- Starting deployment
==> NotasIV:  -- Finished deploying
==> NotasIV: Waiting for SSH to become available...
==> NotasIV: Machine is booted and ready for use!
==> NotasIV: Rsyncing folder: /home/angel/GitHub/NotasIV/ => /vagrant
```

Para acceder a ella, podemos hacerlo con el siguiente comando:

```
$ vagrant ssh
```

Esto funciona porque cuando vagrant crea nuestra máquina, también crea un usuario llamado `vagrant`, y utiliza la llave privada que se encuentra en la ruta que le especificamos con `config.ssh.private_key_path`.

9.3 Aprovisionamiento

Como se ha dicho anteriormente, para aprovisionar la máquina se ha usado ansible, y para decirle qué debe aprovisionar sobre la máquina concretamente he creado un archivo `playbook.yml` en el directorio `provisioning`, que contiene lo siguiente:

```
---
# Como Vagrant nos crea un inventario, aqui podemos poner directamente el nombre que
↪le dimos a la VM.
- hosts: NotasIV
environment:
    PORT: 5000
tasks:
    # Primero con apt vamos a varias dependencias, como pip, make y npm para usar pm2
    - name: Instalar dependencias
      become: true
      apt:
        name:
        - git
        - python3-pip
        - python3-setuptools
        - python-pip
        - nodejs
        - npm
        - make
        state: present
        # Esto ejecuta sudo apt update antes de instalar las dependencias, necesario
        # para que encuentre el paquete python3-pip
        update_cache: true

    # Una vez tenemos npm ahora instalamos pm2 de forma global en el equipo para que
    # cualquier usuario que creemos tenga acceso.
    - name: Instalar pm2 globalmente
      become: true
      npm:
        name: pm2
        global: yes

    # Me creo un usuario angel con una shell de bash. Por defecto le crea un home, no
↪hace
    # falta especificarselo
    - name: Crear usuario angel
      become: true
      user:
        name: angel
        shell: /bin/bash

    # Como queremos configurar este usuario por ssh para acceder a él desde el
↪anfitrion,
    # le mandamos la clave pública que queremos tener autorizada para ese usuario,
    # especificandole la ruta en el anfitrion
    - name: Agregar clave publica para el usuario angel
      become: true
      authorized_key:
        user: angel
        state: present
        key: "{{ lookup('file', '/home/angel/.ssh/id_rsa.pub') }}"

    # Obtenemos el codigo de nuestro repo de GitHub
    - name: Clonar repo de GitHub
      git:
        repo: https://github.com/angelhodar/NotasIV.git
        dest: ~/NotasIV
```

(continues on next page)

(continued from previous page)

```
    # Instalamos las dependencias del proyecto
    - name: Instala librerias necesarias
      pip:
        requirements: ~/NotasIV/requirements.txt
        executable: pip3

    # Usamos la herramienta de construccion para ejecutar la app
    - name: Ejecuta la app
      make:
        chdir: ~/NotasIV
        target: start
        file: ~/NotasIV/Makefile
```

Note: Un detalle importante es que, como explico en el propio playbook al principio, Vagrant nos crea un inventario para ansible en `.vagrant/provisioners/ansible/inventory/vagrant_ansible_inventory` con las maquinas que hayamos definido en el `Vagrantfile`. Como se definió una máquina de nombre NotasIV, podemos ponerla directamente en la clave hosts. Si tuvieramos mas máquinas podriamos agruparlas en un grupo y especificar ese grupo, o simplemente usar el keyword **all** o **default** para ejecutar las tasks del playbook sobre todas las maquinas definidas en el Vagrantfile. Si estuvieramos usando el comando **ansible-playbook** en lugar de vagrant, el inventario por defecto estaría en `/etc/ansible/hosts`.

Una vez tenemos todo listo para aprovisionar la máquina, ejecutamos lo siguiente:

```
$ make provision
```

Lo cual generará un output como el siguiente al ejecutarlo por primera vez:

```
NotasIV: Running ansible-playbook...

PLAY [NotasIV] *****************************************************************

TASK [Gathering Facts] *********************************************************
ok: [NotasIV]

TASK [Instalar dependencias] ***************************************************
changed: [NotasIV]

TASK [Instalar pm2 globalmente] ************************************************
changed: [NotasIV]

TASK [Crear usuario angel] *****************************************************
changed: [NotasIV]

TASK [Agregar clave publica para el usuario angel] *****************************
changed: [NotasIV]

TASK [Clonar repo de GitHub] ***************************************************
changed: [NotasIV]

TASK [Instala librerias necesarias]␣
↪*********************************************************
changed: [NotasIV]
```

(continues on next page)

(continued from previous page)

```
TASK [Ejecuta la app] ****************************************************
changed: [NotasIV]

PLAY RECAP ***************************************************************
NotasIV  : ok=6  changed=7  unreachable=0  failed=0  skipped=0  rescued=0  ignored=0
```

Las tareas marcadas con **changed** viene a decir que esa tarea se ha realizado y ha cambiado el estado de la máquina. Si por el contrario pusiera **ok**, significaría que esa tarea ya ha sido ejecutada y tenemos el sistema con el estado requerido para esa tarea, por lo que no es necesario ejecutarla.

Veamos a grandes rasgos qué hace nuestro playbook:

1. Usando el modulo **apt** de ansible, instala y actualiza las dependencias necesarias para crear el entorno necesario para ejecutar la app.
2. Usando el módulo **npm** instalamos pm2, necesario para tener control sobre la ejecución de nuestra app
3. Creamos un usuario llamado *angel* con el módulo **user**, asignándole un shell de bash en lugar de sh que es el que viene por defecto.
4. Al usuario le asignamos la llave pública del par que vamos a usar para conectarnos a la máquina con ese usuario.
5. Clonamos el repo de GitHub.
6. Instalamos las librerias de python necesarias para nuestro proyecto con pip.
7. Arrancamos el servicio con nuestra herramienta de construcción.

Para conectarnos con ssh a la máquina usando el usuario *angel* que hemos creado en el aprovisionamiento, debemos hacerlo con el comando **ssh** en lugar de **vagrant ssh**, usando el puerto 22 para acceder y la IP pública que nos asigna Azure a nuestra máquina, que en este caso es **52.236.139.44**

```
$ ssh angel@52.236.139.44 -p 22
```

Note: Suponemos que tenemos la llave privada asociada a ese usuario en `~/.ssh` en nuestra máquina, de lo contrario deberiamos de especificarselo al comando ssh con la opción **-i**.

Python Module Index

n

t

Index